Data Representation in Computers

How Computers Represent Numbers, Text, Images, and Sound

Artem Kovera

Last updated 2021-03-01

Table of Contents

Why learning how computers represent data is important

Obviously, no one can become a good player without understanding the rules. Low-level knowledge and an understanding of the underlying principles is vital in any field and it is also absolutely essential for the computer world.

At the low level of abstraction, computers just manipulate data in memory. An understanding of computer data representation is one of the most important pieces of knowledge every good IT-specialist should have. Everything else is pretty much built on it. If a programmer does not understand the principles of how computers represent different types of data, such a programmer is obviously prone to make more mistakes in his or her code, or, in some situations, the programmer will not be able to work at all.

A good understanding of how computers store and manipulate data – especially numeric information – will help the programmer produce much more efficient code. An understanding of number representations in computers is vital for scientific computing, computer graphics, data science, and many other very important fields.

Also, other than just performing computations, computers must communicate with people. Computers must accept some input from people and deliver information back to people. If people could communicate with computers only just by using numerical information as input and output, computers would not be much different from pocket calculators. In order to make communication between computers and people much reacher and easier, computers must be able to accept, process, and deliver textual, audio, and image information. And we as computer programmers or other computer professionals must know at least the fundamental principles of how these various types of information are represented in computers.

In addition to all of that, nowadays, there are lots of hackers who will exploit any vulnerability for accessing target computer systems. So, a good programmer must also be able to produce secure code. And a good

understanding of computer data representation is one of the keys to writing secure programs.

I hope this e-book will help you take the first steps towards acquiring fundamental low-level knowledge of how computers represent different types of data.

Binary, Octal, and Hexadecimal number systems

There is an infinite number of possible number systems. We, as humans, usually operate with the decimal (10 digits) number system. The decimal system is used around the power of 10.

For example, the number 1257_{10} can be represented as:
$(1 * 10^3) + (2 * 10^2) + (5 * 10^1) + (7 * 10^0) = 1257_{10}$.

Or, the number 10803_{10} can be represented as:
$(1 * 10^4) + (0 * 10^3) + (8 * 10^2) + (0 * 10^1) + (3 * 10^0) = 10803_{10}$.

To avoid confusion, the suffix on the right – in the above examples 10 – indicates the base of the number system.

Using 10 digits is very intuitive for us, but computers represent information in a different way at the low level.

As you probably know, computers use two-valued signals for storing any data and performing computations. So, computers use binary numbers (2 digits: 0 and 1). In computers, any information – such as executable, text, images, sound, or video files – is stored and processed exclusively by using only two fundamental values: 1 and 0.

A single binary digit is known as a **bit**. It can represent the numerical values 0 and 1. Also, a single bit can represent the logical values FALSE and TRUE, which are very important entities in computer programming.

Any binary number can be represented as a sum of powers of 2

For example, $1010_2 = 8 + 0 + 2 + 0 = 10_{10}$

$0 * 2^0 = 0$
$1 * 2^1 = 2$
$0 * 2^2 = 0$
$1 * 2^3 = 8$
10_{10}

Or, for example, $101101_2 = 32 + 0 + 8 + 4 + 0 + 1 = 45_{10}$

$1 * 2^0 = 1$
$0 * 2^1 = 0$
$1 * 2^2 = 4$
$1 * 2^3 = 8$
$0 * 2^4 = 0$
$1 * 2^5 = 32$
45_{10}

Just like with decimal numbers, we can do arithmetic operations on binary numbers.

In decimal addition, we add up each consecutive digit from right to left, and if the result is greater than 9, we carry the 1. Similarly, we add together each bit in a binary number, and if the result is greater than 1, we carry the 1.

When the result of addition doesn't fit in the number of bits that we have, it results in *overflow*, which happens whenever we have a carry from the rightmost bit (the most significant bit).

It's very useful to remember by hart first values of power of 2:

$2^2 = 4$
$2^3 = 8$
$2^4 = 16$
$2^5 = 32$
$2^6 = 64$
$2^7 = 128$
$2^8 = 256$
$2^9 = 512$
$2^{10} = 1024$
$2^{11} = 2048$
$2^{12} = 4096$

2^{13} = 8192
2^{14} = 16384
2^{15} = 32768
2^{16} = 65536

It is also helpful to know that 2^{10} (1024) is close to 10^3 (1000). It allows us to approximately convert large numbers between binary and decimal systems. For example, if we want to know how many bits are needed to represent one billion (10^9) in binary, we can determine that it is some number between 2^{29} and 2^{30}, so we need at least 30 bits to represent this number in binary.

Now, let's briefly discuss octal (base 8) and hexadecimal (base 16) number systems. They are also important in the computer world.

Big binary numbers are often too overwhelming to work with. Converting decimal numbers to binary may be tricky. At the same time, it is easy to convert octal or hexadecimal to binary since it can be performed one hexadecimal or octal digit at a time. For these reasons, octal and hexadecimal number systems are often used in many situations in the computer world.

In the octal number system, the following digits are used: 0, 1, 2, 3, 4, 5, 6, 7.

The 8 in decimal corresponds to the 10 in octal. The octal number system revolves around the power of 8.

For example 1705_8 =

$5 * 8^0 = 40_{10}$
$0 * 8^1 = 0$
$7 * 8^2 = 448_{10}$
$1 * 8^3 = 512_{10}$
1000_{10}

Each group of consecutive three bits can be represented as one octal digit, which is convenient.

For example, $100_2 = 4$; $111_2 = 7$; $011_2 = 3$
So, the binary number 100 111 011 corresponds to 473 in octal.

The hexadecimal number system is even more prevalent in the computer world than the octal number system. For example, virtual memory addresses are usually represented in hexadecimal. There are a lot of other examples. This number system is used around the power of 16 and has 16 digits.

Since the decimal number system has only 10 digits, the hexadecimal system uses six additional digits: **A, B, C, D, E,** and **F**.

It's very convenient that all possible values of one hex digit can be represented by 4 binary digits and vice versa:

Hexadecimal	Binary
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

For example, the 32-bit binary number

1010 1110 0010 1011 1010 0000 1011 0100 can by represented as

A E 2 D A 0 B 4 in hexadecimal

So, when we convert a binary number to hexadecimal, we split this number into groups of 4. Then we convert each group to a hexadecimal digit.

If the number of bits is not a multiple of 4, we should make the leftmost group of bits be the one with fewer than 4 bits.

For example, let's take this binary number: **10101101010** and convert it to hexadecimal. This number has 11 bits, and it is not a multiple of 4. So, we divide this number into 3 groups in the following way: **101 0110 1010**. Then we convert each group to hexadecimal: 56A.

A byte has 8 bits (2 * 4), so any byte can be represented as a pair of hexadecimal values, which is extremely convenient.

In high-level languages such as C/C++, hexadecimal values are denoted by a leading "0x". For example, a hexadecimal number AE2DA0B4 from the previous example is written as 0xAE2DA0B4 or 0xae2da0b4 in these

languages. Using this prefix with hexadecimal also applies to other situations.

Here is a table of the first 32 numbers written in different number systems:

Decimal	Binary	Octal	Hexadecimal
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10
17	10001	21	11
18	10010	22	12
19	10011	23	13
20	10100	24	14
21	10101	25	15
22	10110	26	16
23	10111	27	17
24	11000	30	18
25	11001	31	19
26	11010	32	1A
27	11011	33	1B
28	11100	34	1C
29	11101	35	1D
30	11110	36	1E
31	11111	37	1F
32	100000	40	20

A few words about computer memory organization

A bit is the most fundamental and smallest container of computer memory. But, usually, we as programmers work with other computer memory containers. A **byte** is the second most fundamental computer memory container. In most computer systems, a byte is a group of 8 bits. An 8-bit byte is also known as an *octet*. In this book, I will always imply by a byte only an 8-bit container.

The range of possible values of a single unsigned byte is from 00000000 to 11111111 in binary, or from 0 to 255 in decimal, or from 00 to FF in hexadecimal. So, there are 256 different possible values each byte can have.

A half-byte 4-bit container is known as a *nibble*. So, each byte consists of two nibbles. Each nibble corresponds to one hexadecimal digit. The range of possible values of a single unsigned nibble is from 0000 to 1111 in binary, or from 0 to 127 in decimal, or from 0 to F in hexadecimal.

Oftentimes memory containers are represented using larger computer memory containers. For example, in the C/C++ programming languages there are following fundamental integer types, most of which occupy more than one byte:

char – 1 byte
short – 2 bytes
int – 4 bytes
long – 4 or 8 bytes depending on the architecture
long long – 8 bytes

By the way, the low-level nature of the C and C++ languages makes them ideal for demonstrating and learning how different types of data are represented in computer memory.

Memory in most computer systems is byte addressable. This means that each byte has its own number in memory. The set of all possible byte addresses is known as *virtual address space*. The dominant architecture of

laptop, desktop, and server computers (X86/X86_64) belongs to the computer systems with byte-addressable memory.

In the case of more-than-one-byte memory containers, the address of the memory container corresponds to its smallest byte address.

Also, when we use more-than-one-byte integer containers, we should agree on the order of memory addresses. For example let's write the decimal number 2222222222 as a binary number: **10000100 01110100 01101011 10001110**. In this number, the most significant byte is **10000100** and the least significant one is **10001110.** For example, we have the following memory addresses: 100, 101, 102, 103. There are two major ways how we can write this number in memory. In one option, the least significant byte comes first and the most significant byte comes last. And the second option is just the opposite.

So, the first way to place this number in memory:

100 - **10001110** (least significant byte)
101 - **01101011**
102 - **01110100**
103 - **10000100** (most significant byte)

And the second way:

100 - **10000100** (most significant byte)
101 - **01110100**
102 - **01101011**
103 - **10001110** (least significant byte)

When the least significant byte comes first, such an architecture is known as *little-endian*. For example, the aforementioned x86/x86_64 architecture is little-endian. On the contrary, the most significant byte comes first in the *big-endian* architecture.

Some families of processors are *bi-endian*, which means they can operate in both modes. One of such architectures is ARM processors. However, in practice, ARM processors usually operate as little-endian machines.

(Every computer has a *word size*, which is a size of pointer data. For example, currently most laptops and desktop computers have processors belonging to the x86-64 architecture family. Their word size is 64 bits or 8 bytes, and their virtual address space has 2^{64} possible addresses.)

Representing integer numbers in computers

Integer numbers can be either signed or unsigned. In the computer world, unsigned integer numbers are used only to represent positive values. Signed integers can represent both negative and positive values.

Representing unsigned integer numbers is super simple: we just place the binary number in memory padding with zeros all vacant bits if any. Also, we should consider the endiance of the machine to place the bytes in the right order. In reality, all this happens automatically luckily for us.

For example, if we place 2 in a one-byte variable, we have:
00000010.

If we place 2 in a two-byte variable, we have:
00000000 00000010.

If we place 2 in a four-byte variable, we have:
00000000 00000000 00000000 00000010.

The ranges of possible values for unsigned integer variables:

1-byte (char)	**from 0 to $2^8 - 1$ (255)**
2-byte (short)	**from 0 to $2^{16} - 1$ (65535)**
4-byte (int or long)	**from 0 to $2^{32} - 1$ (4294967295)**
8-byte (long or long long)	**from 0 to $2^{64} - 1$ (very big number)**

We can get unexpected results doing integer arithmetic For example, suppose we add 200 to a one-byte unsigned integer variable, which holds 200 already. 200 in binary is 11001000. So, we add 11001000 and 11001000 and we get 110010000 – and this is 9 bits, not 8! A one-byte variable consists of 8 bits, so in this situation the most significant bit in the result is thrown away. So, instead of 110010000, actually, we have 10010000, which is 144 in decimal. In such cases, we have a situation that is known as **overflow** or **wraparound**. For another example, let's suppose that we add 128 to a one-byte variable, which already holds 128. 128 in binary is 10000000. So, we add 10000000 and 10000000 and get 100000000 – and this is 9 bits as in the previous example. So, the leading

bit is discarded and we have 00000000. So, we added 128 and 128 and got 0 back!

Also, overflow may occur when we multiply two numbers. In this case, more than one digit can be discarded. For example, when we multiply two one-byte integer variables that store 255, we get 11111111 * 11111111 = **11111110 00000001** in binary, which is 65025 in decimal. If we store the result into a one-byte variable, the most significant byte of the result (11111110) is discarded, so we end up with 00000001, which is one!

Integer overflow bugs may lead to fatal errors and create serious vulnerabilities in computer programs. Because of this, computer programmers should be acutely aware of unintentional integer overflow possibility, choose their integer variable types carefully, and use the variables appropriately to avoid overflow.

However, overflow may be used intentionally in some situations. In C and C++, wraparound behavior using unsigned integer numbers is well-defined, so it can be used for some purposes intentionally. Even the term *overflow* is not really correctlly applied to unsigned integer arithmetic in C and C++; instead, the terms *wraparound behavior* or *wraparound operation* should be used in this context.

In addition to just positive unsigned numbers, we should have a method to represent negative numbers. For this purpose – as I've already mentioned – signed numbers are used, which can be positive and negative.

There are three main methods to represent negative numbers. Two methods to represent negative numbers are rarely used and include the so-called signed magnitude notation and the one's complement notation. In most computer systems, the so-called **two's complement notation** is used to represent negative numbers.

In two's complement notation, the most significant bit is always a zero for all positive numbers. For example, those are positive one-byte numbers:

00000001 1_{10}
01001000 72_{10}
01111111 127_{10}

So, positive numbers in two's complement notation are pretty straightforward.

In order to make a negative number in two's complement, we need to invert all the bits of the corresponding positive number and then add one to the result.

For example, to make a negative one for a one-byte variable, we invert all the bits in a positive one number (00000001). As a result, we get 11111110. Then we add one and get 11111111.

For another example, let's convert a positive one-byte 2 (00000010) to negative. In the first step, we invert all the bits: 11111101. Then, we add 1: 11111101 + 1 = 11111110.

Finally, let's convert a positive one-byte 127 (01111111) to a negative value. In the first step as usual, we flip the bits: 10000000. Then, we add 1: 10000000 + 1 = 10000001.

If we had a 4-bit integer variable, we would have the following sequence of possible values in the two's complement notation:

0000	**0**
0001	**1**
0010	**2**
0011	**3**
0100	**4**
0101	**5**
0110	**6**
0111	**7**
1111	**-1**
1110	**-2**
1101	**-3**
1100	**-4**
1011	**-5**
1010	**-6**
1001	**-7**
1000	**-8**

As you can see, we have one additional negative number, which is -8 in this example.

There are a few special patterns in two's complement notation no matter how many bytes the variable has. The first pattern is when all the bits are equal to zero: in this situation, we have a zero. The second one is when all the bits are equal to 1: in this situation, we have a -1. The third pattern is when the most significant bit is equal to 1 and all the other bits are equal to 0: in this situation, we have the smallest value the integer variable can represent.

The ranges of values for different signed integer types:

1-byte variable **-128 to 127**
2-byte variable **-32768 to 32767**
4-byte variable **-2147483648 to 2147483647**
8-byte variable **-9223372036854775808 to 9223372036854775807**

When dealing with signed numbers, we should be concerned with possible overflow happening. In two's complement, we can get a negative number by adding two positive ones, and we can get a positive number by adding two negative ones. Even worse, in the C/C++ programming languages, when overflow occurs in signed variables, we end up with *undefined behavior*, which means that the C or C++ standards do not specify what happens exactly, so different compilers can do different things in the same situation, which is very dangerous.

In computers, integer arithmetic is associative. For example, we can add these three values: 50, 100, 200 and store the result into an unsigned one-byte variable. By adding the values, we get 350 or 101011110 in binary, which is 9 bits. The most significant bit is discarded so we and up with 01011110 or 94. If we change the order of addition, for example, 100, 200, 50, we will again and up with the same value in the result. So, the result is incorrect, but at least it's consistent. On the other hand, arithmetic with floating-point numbers is not necessarily associative, and we will be discussing it in the next chapter.

In C and C++, we can change a type of integer variable into another type. For example, we can cast an **int** variable into a **short** variable or vice versa. In these cases, we should be concerned with possible information loss if we cast a variable into a smaller type.

Division and multiplication operations are among the most time-consuming operations in computer processors. The division is especially costly. If, for example, an addition operation takes around one clock cycle, the division may take around 30 clock cycles. It is worth pointing out that division and multiplication by powers of 2 can often be replaced by just shifting the bits to the left if we perform multiplication or to the right if we perform division. For example, if we want to divide the number 01000000, we just shift the 1 to one position to the right: 00100000. And if we want to multiply the result by 4, we shift the 1 to two positions to the left: 10000000.

Since computer integer arithmetic is associative and oftentimes some arithmetic operations can be replaced with other ones, compilers usually perform a whole lot of optimization on integer arithmetic. For example, they replace division operations with combinations of shifting and some other types of operations. We can observe various compiler optimizations of integer arithmetic in disassembly, for example.

Representing floating-point numbers

So far in this book, we have been dealing exclusively with representing whole numbers (integers). But we know that the world is full of numbers that are not necessarily whole. Those numbers are fractions (such as 1/3 or 0.47) or irrational numbers (such as π or √2).

All irrational numbers cannot be represented exactly in any numeral system. So, we can only represent irrational numbers as fractions, always losing some degree of precision in this process.

Let's discuss how decimal and binary fractions work. Everyone knows (hopefully) that we cannot represent exactly some fractions in decimal, for example, 1/3. Similarly, we cannot represent exactly many fractions in binary notation. The only binary fractions that can be represented exactly are those where the denominator is a power of 2.

For example, the following numbers can be represented exactly in binary notation because their denominators are powers of 2:

1/2,
1/4,
1/8,
1/16,
3/4 (1/4 * 3),
5/8 (1/8 * 5).
11/16 (1/16 * 11)

But such fractions as, for instance, 1/3, 2/5, 7/9 cannot be represented exactly in binary.

We also should understand that we can represent certain numbers in decimal – for example, 2/5 (0,4) – but we cannot represent the same numbers exactly in binary.

In fact, the only decimal fractions that can be represented exactly in binary are some of those that end in 5, for example, 0.5(1/2), 0.125(1/4), 0.625(5/8), 0.75(3/4).

For a better understanding of binary fractions, consider the following:

0.1_2	=	0.5_{10}	(1/2)
0.01_2	=	0.25_{10}	(0.5/2)
0.001_2	=	0.125_{10}	(0.25/2)
0.0001_2	=	0.0625_{10}	(0.125/2)
0.00001_2	=	0.03125_{10}	(0.0625/2)

You can see the pattern: each lower fraction is exactly twice as small as the upper fraction. Now, by combining such fractions, we can produce other fractions.

For instance:

0.101_2 $(0.1_2+0.001_2)$ = 0.625_{10} $(0.5_{10}+0.125_{10})$

0.01101_2 $(0.01_2+0.001_2+0.00001_2)$ = 0.40625_{10} $(0.25_{10}+0.125_{10}+0.03125_{10})$

In most general-purpose computers, the method to represent fractions is using the so-called **floating-point representation**.

Each floating-point number consists of three parts: a *sign bit*, a *binary fraction* (also known as a *mantissa*), and a *binary exponent*. The floating-point number is positive if the sign of this number is a zero and it is negative when the sign is a one. The mantissa determines the precision of the floating-point number. The exponent determines the range.

In order to understand floating-point numbers, we need to understand a scientific notation and its normalized form. In scientific notation, any nonzero number can be written as the product of a mantissa (fraction) and an exponent. In the normalized form, the absolute value of the mantissa is at least 1.0 but less than 10.0. Usually, this notation is used for representing very large or very small numbers. For example, the mass of a stationary electron written in a normalized scientific notation is about $9.1 * 10^{-31}$.

In case you forgot, I'll remind you that using negative powers allows us to represent fractions. Let's first take a look at negative powers in decimal numbers. For example, $0.1_{10} = 10_{10}^{-1} = 1/10_{10}^{1}$ and $0.01_{10} = 10_{10}^{-2} = 1/10_{10}^{2}$.

In the same fashion, we can work with binary numbers too. For example $0.1_2 = 10_2^{-1} = 1/10_2^1$ and $0.01_2 = 10_2^{-2} = 1/10_2^2$.

So, binary nonzero numbers can also be written in a normalized form. In a normalized form of a binary number, the leading 1 is always presented. In floating-point notation, in order to use more bits for getting higher precision, the mantissa doesn't store the leading 1, but it is used in calculations implicitly. This method of a mantissa representation is sometimes called an implied leading 1 representation.

If we have a positive exponent, which means our number is bigger than 1, we have to move the point to the right. On the other hand, if we have a negative exponent, we have to move the point to the left and pad the vacant positions with zeroes, hence the name *floating-point*.

It is important to understand that floating-point numbers are just an approximation to true real numbers. There is always an infinite amount of real numbers between any two mathematical real values (no matter how close they are to each other), but it is not the case with floating-point numbers. For example, in a single-precision floating-point representation, there are about 16 million possible values between 1.0 and 2.0.

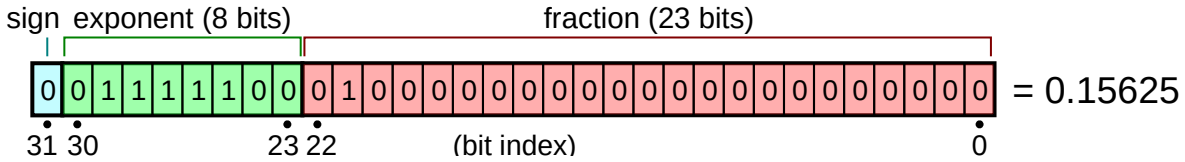
So, it's impossible to represent certain floating-point numbers exactly in a computer, and it leads to round-off errors in calculations. Such errors are usually minuscule and seem to be harmless. But with the increasing number of calculations, these errors may accumulate and eventually significantly compromise the results. Moreover, such errors can lead to many bugs if the programmer is not aware of floating-point number properties, at least in general. One possible example of such a bug, I will show at the end of this chapter.

We could use different numbers of bits for floating-point number parts on different machines. But in this situation, the computer programs would be totally incompatible with other computers. So, we need an industry-wide standard for floating-point representation. And this standard exists and is called the **IEEE 754 standard.**

IEEE 754 defines three binary floating-point basic formats commonly known as **single precision**, **double precision**, and **quadruple precision**,

which are encoded with 32, 64, and 128 bits respectfully. In most programming languages, the single precision and double precision formats are used most often.

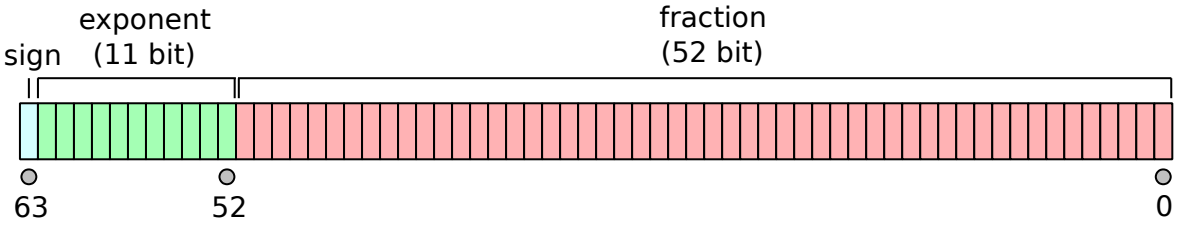
Single precision has one bit for the sign (sign bit), 8 bits for the exponent, and 23 bits for the mantissa:



Source

The 24 mantissa (23 actual + 1 implied) bits of a single precision floating-point number allows us to accurately represent only 7 and sometimes 8 decimal digits. So, for example, the decimal numbers 1234567 or 123.0625 can be exactly represented in a single precision floating-point number. But all decimal numbers consisting of more than 8 digits cannot be represented exactly in single precision. The 8-bit exponent of a single precision floating-point number allows representing numbers in a range roughly from 10^{-38} to 10^{38}.

Double precision has one bit for the sign, 11 bits for the exponent, and 52 bits for the mantissa:

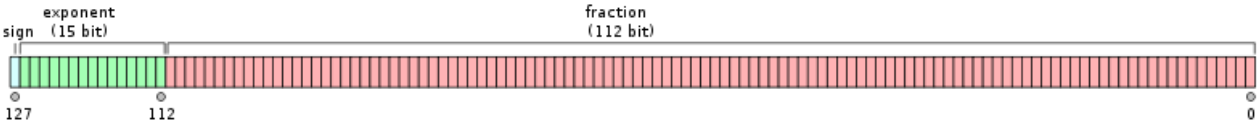


Source

Double precision numbers are more accurate than single precision numbers. So in many situations, it’s recommended to use double precision numbers. The 53 bits mantissa allows us to accurately represent 15-17

decimal digits. The 8-bit exponent of a double precision floating-point number allows representing numbers in a range roughly from 10^{-308} to 10^{308}.

Quadruple precision has one bit for the sign, 15 bits for the exponent, and 112 bits for the mantissa; therefore it requires 128 bits in total. Naturally, quadruple precision floating-point numbers are more accurate and allow for more range of values than double precision numbers.



Source

In the C/C++ programming languages, the single precision format corresponds to the **float** basic data type, the double precision format corresponds to the **double** basic data type, and the quadruple precision most often corresponds to the **long double** basic data type.

In addition to the 3 basic floating-point formats, the IEEE 754 specifies a few other floating-point formats. One of these additional formats is encoded with 16 bits and is known as **half precision** or **base16**.

We need a method for representing negative exponents in the exponent part of floating-point numbers. There is a nifty little trick for that in IEEE 754. A computer adds an actual value of the exponent to a special value known as **bias**. For the single precision format, the bias is 127, or encoded in binary it is 01111111.

For example, if we need to represent the exponent equal to **2** in the single precision format, we add 127 to 2. Or in binary, we have 01111111 + 00000010 = **10000001**. So, we store 1000 0001 in the exponent part of the number.

Conversely, if we need to represent the exponent equal to **-2**, we subtract 2 from 127, or in other words, we add -2 to 127. Therefore in binary, we

have 01111111 – 00000010 = **01111101**. So, we store 01111101 in the exponent part of the single precision floating-point number.

Let's go over several concrete examples of floating-point numbers. In the examples, I will be using a single precision notation.

Let's begin with **1.0**. It's a positive number, so the sign bit is 0. We can view 1.0 as $1.0 * 2^0$. The exponent is 0, but we have to add 127 to it, so the actual exponent is 01111111. Since we don't store the leading 1 in the actual mantissa the mantissa is just all zeros: 00000000000000000000000. As a result, we have **0 01111111 00000000000000000000000**.

Now, let's take **2.0**. We can view 2.0 as $1.0 * 2^1$. It's a positive, so the sign bit is 0. The exponent is 1, but after adding 127, the actual exponent becomes 10000000. Since we don't store the leading 1 in the actual mantissa the mantissa consists only of zeros: 00000000000000000000000, just like in the previous example. So, as a result, we have **0 10000000 00000000000000000000000**.

Now, let's take **1.5**. We can view this number as $1.5 * 2^0$. Since it is a positive number, the sign bit is 0. The exponent is 0, but, as always, we have to add 127 to it, so the actual exponent is 01111111. If we write 1.5 in binary, it becomes 1.1. Since we don't store the leading 1 in the actual mantissa the mantissa is 1 followed by zeros: 10000000000000000000000. So, in the end, we have: **0 01111111 10000000000000000000000**.

Now, let's take **-5**. Since it's a negative number the sign bit is 1. We can view 5 as $1.25 * 2^2$. The exponent is 2, but, as always, we have to add 127 to it, so the actual exponent becomes 10000001. When we write 1.25 in binary, it becomes 1.01. Since we never store the leading 1 in the actual mantissa the mantissa becomes 01 followed by zeros: 01000000000000000000000. So, in the end we have: **1 10000001 01000000000000000000000**.

Now, let's observe **0.3125**. Since it's a positive number the sign bit is 0. we can view this number as $1.25 * 2^{-2}$. The exponent is -2, but we have to add 127 to it, so the actual exponent becomes 125 or 01111101. Decimal 1.25 in binary is 1.01. Like in the previous example, the mantissa becomes 01

followed by zeros: 01000000000000000000000. So, in the end, we have: **0 01111101 01000000000000000000000**.

There are several special values in floating-point numbers: Zero, Not a Number (NaN), Plus Infinity, and Minis Infinity.

Zero is just a zero. Zero is represented with all zeros in the exponent and all zeros in the mantissa. Actually, the IEEE 754 standard has two versions of zero: plus zero and minus zero. In most situations, the distinction between +0 and -0 doesn't matter.

Not a Number (NaN) occurs at cases like division zero by zero or taking the square root of a negative number. All arithmetic operations that involve NaN as one of their operands result in NaN. So, if NaN once occurs, it propagates to the end of the calculations. Not a Number is represented when all bits of the exponent are ones, and at least one bit in the mantissa is also one. For example, here is a possible representation of a single precision NaN: **0 11111111 10000000000000000000000**.

Plus Infinity occurs, for example, when we divide a positive number by zero. The form of the plus infinity representation is the following: the sign bit is zero, the exponent's bits are ones and the mantissa bits are zeros. For example, here is the representation of a single precision plus infinity:
0 11111111 00000000000000000000000. Not a Number (NaN) occurs at cases like division zero by zero or taking the square root of a negative number. All arithmetic operations that involve NaN as one of their operands result in NaN. So, if NaN once occurs, it propagates to the end of the calculations. Not a Number is represented when all bits of the exponent are ones, and at least one bit in the mantissa is also one. For example, here is a possible representation of a single precision NaN: 0 11111111 10000000000000000000000.

Minus Infinity occurs when we divide a negative number by zero. The representation of minus infinity is exactly the same as plus infinity except for the sign bit. For example, here is the representation of a single precision minus infinity: **1 11111111 00000000000000000000000**.

Infinity may also occur when we multiply very big numbers. When we divide a number by infinity, we get zero. When we divide infinity by a

number (no matter how big), we get infinity. When we multiply infinity by infinity, we get infinity. When we divide infinity by infinity or subtract infinity from infinity, we get NaN.

The floating-point number is in a **denormalized form** when the exponent field is all zeros and the mantissa is not all zeros. The denormalized form occurs when we get a number which is very close to 0.0. This form allows us to represent numbers much closer to 0.0 but with less precision than would be possible just by using negative exponents. For example, here is a single precision number in a denormalized form:
0 00000000 01000100000000000110000.

Also, a condition known as **underflow** can occur in floating-point arithmetic. When underflow occurs, it means that we have a so small number that even a denormalized form cannot represent it. And floating-point underflow results in a zero.

It is important to understand that possible values of floating-point numbers are not evenly distributed: they are denser near the zero.

Floating-point numbers do not necessarily obey some common mathematical properties such as associativity and distributivity due to the finite precision of the representation.

Floating-point arithmetic instructions in a processor are considerably slower than integer instructions Moreover, due to nonassociativity and nondistributivity of floating-point arithmetic, compilers do not perform many optimizations on floating-point arithmetic as they do on integer arithmetic. All of that leads to slower computer performance.

However, modern processors can exploit data-level parallelism and have extended floating-point registers, using which allows performing a single instruction on several floating-point or integer numbers at a time. This is known as the *single instruction, multiple data (SIMD)* paradigm and it can significantly improve the performance of floating-point arithmetic.

Since the number of bits in a floating-point representation is finite, oftentimes some sort of rounding is performed. Rounding may be implemented in different ways. For example, the number 1.5 we can round

either to 1 or 2. The IEEE 754 standard defines 4 possible methods for rounding floating-point numbers.

Round-to-even (also called **round-to-nearest**) is the default mode. In this mode, the floating-point number is always rounded to its nearest match. For example, by adopting this approach, the values 1.2 or 1.4 would be rounded to 1, and the values 1.6 or 1.9 to 2. When the number is exactly halfway between two possible results – for instance, 1.5, 2.5, or 10.5 – the round-to-even method rounds the value such that the least significant digit of the result is even. In this case, for example, the values 1.5 and 2.5 would be both rounded to 2. For example, the GCC compiler uses this mode as a default.

In addition to the default round-to-even mode, there are 3 other modes for rounding floating-point numbers.

Round toward plus Infinity. In this mode, the floating-point number is rounded to the smallest representable value which is greater than the result.

Round toward minus Infinity. The floating-point number is rounded to the largest representable value which is greater than the result.

Round toward zero. If the result is negative the floating-point number is rounded up; if the result is positive, the number is rounded down.

The non-precise nature of floating-point representation may lead to various bugs in software. Programmers should be especially careful when comparing floating-point numbers. Let me illustrate one possible example of floating-point comparison that may lead to erroneous behavior of the program. Here is the code in C:

```
float v1 = 1.0;
float v2 = 0.0;

for (int i = 0; i < 10; i++)
{
        v2 += 0.1;
}
```

```
if (v1 == v2)
{
        printf(“They’re equal”);
        … (code that is supposed to be executed)
}
else
{
       printf(“They are not equal”);
       … (other code)
}
```

In this little example, we have to floating-point variables: v1 and v2. Initially v1 is equal to 1 and v2 is equal to 0. We add 0.1 10 times to v2 in a loop. Then we perform a comparison between v1 and v2. A naive beginner programmer might think that v2 is equal to 1 after the loop, so v1 and v2 are equal. But actually, v2 is not equal to 1 after the loop. It’s slightly bigger, because we cannot represent the decimal value 0.1 exactly in binary floating notation and it’s represented by the actual value that is slightly bigger than 0.1. So, v2 and v1 are not equal. Many similar bugs are also possible. So, programmers must be careful with floating-point comparison and other operations involving floating-point numbers.

Binary-Coded Decimal (BCD) Representation

In addition to regular integer and floating-point binary number representations we've just discussed, sometimes some other number representation methods are used in computer systems.

One such alternative is the so-called binary-coded decimal representation (BCD), in which we use 4 bits to represent each decimal digit (0000 (0), 0001 (1), ... 1000 (8), 1001 (9)).

So, each byte holds two BCD digits. For example, the decimal 10 is encoded as 0001 0000. In the BCD, each byte can represent 100 different numbers. Therefore, this method requires more computer memory than regular binary numbers. The BCD allows easy conversion between binary and decimal. The BCD is mainly used for eliminating conversion errors between binary and decimal fractional numbers.

The use of BCD is more common in simpler electronic devices rather than in computers. For example, BCD is very common in devices that display decimal digits, such as pocket calculators or digital clocks.

Introduction to Representing Text

Computers can understand only 1s and 0s. But how can other than numeric types of data be represented in computers?

It's possible by using various types of **encoding**. In the world of computers, encoding is the process of converting data into a format suitable for computers to store and process, in other words, in sequences of 1s and 0s. There are many types of encoding used for different types of data, including images, audio, video, and characters.

For representing textual data, there are multiple encoding methods. But probably the most important one is the **American Standard Code for Information Interchange (ASCII).** This encoding uses 7 bits to represent each symbol.

Since this encoding uses 7 bits, there are 128 different symbols represented by it ($2^7 = 128$). It is enough to represent capital and lower-case English letters, decimal digits, different punctuation symbols, plus some other special symbols (so-called control characters).

For example, in ASCII, the character "A" is encoded as **65** in decimal, or **41** in hex, or **0100 0001** in binary; and the following character "B" is encoded as **66**, **42**, or **0100 0010** respectfully.

In the C and C++ programming languages, ASCII is the default text encoding. Computer programs must know where the textual information ends so as not to interpret the following code as text. For the purpose of determining the end of textual information, different control characters are used. In C and C++, the text terminating symbol is the Null ASCII symbol, which is encoded as **0000 0000** in binary.

Other control characters are used for different purposes. For example, symbols encoded as **0000 0101** and **0000 1011** are used for horizontal and vertical tabulations. The functions of some control symbols may vary on various platforms.

Also, it’s worth knowing that 10 decimal digits in ASCII are encoded as follows:

0011 0000 – 0
0011 0001 – 1
0011 0010 – 2
0011 0011 – 3
0011 0100 – 4
0011 0101 – 5
0011 0110 – 6
0011 0111 – 7
0011 1000 – 8
0011 1001 – 9

You can see that these values differ in only their rightmost 4 bits and the values of these bits coincide with the values of the corresponding decimal digits. It simplifies conversion between text numbers and numeric values of those numbers.

But having the 128 ASCII symbols is certainly not enough to represent letters from other languages. For this reason, multiple extended ASCII standards have been developed, which use 8 bits for encoding one symbol. Therefore, each extended ASCII encoding allows representing 256 individual symbols. And each such method encodes the 128 symbols exactly as it does the ASCII. So, all these standards are backward compatible with the ASCII.

Obviously, using different extended ASCII standards does not allow us to solve all the problems with a computer text representation. One problem which may arise is that several different languages can be used in a given piece of text at the same time, for example, English, Greek, and Russian. Another problem is that transmuting text documents between different countries may sometimes be very difficult and certainly requires additional effort when multiple encodings are used in different countries. To make the matter even worse, some European languages have several different extended ASCII encodings. In addition to all of that, there are certain languages that use more than 128 or 256 symbols to represent text – sometimes way more – so, they cannot be encoded at all by using extended one-byte ASCII encodings.

To address all such problems, a new standard called **Unicode** was invented in the early 1990s. Today, Unicode forms the foundation for the representation of various languages and symbols in all major operating systems, user applications, and the Internet.

It is important to understand that Unicode is not a character encoding. Actually, Unicode defines several different character encodings. Currently, the most popular Unicode encodings are UTF-8 and UTF-16. Some Unicode encodings are now obsolete, like UCS-2.

The Unicode standard defines a set of characters, which can be encoded by its different encodings, and provides a unique number for each such character.

In Unicode, a character number is called a *code point*. Unicode code points are usually denoted as **U+hhhh(h)**, where *hhhh(h)* is 4 or 5 hexadecimal values. In the Unicode point code notation, the ASCII symbols are zero-extended to 4 hexadecimal values. For example, the Unicode symbol for “A” is denoted as U+0041 (“A” is encoded as hexadecimal 41 in ASCII).

It is important to understand that Unicode code points are not necessarily the same as their actual codes in different Unicode encodings. For example, the code point for the capital Cyrillic letter “Д” is U+0414. But, this character code is 0xD094 in the UTF-8 encoding, 0x0414 in the UTF-16 (in this case the codes coincide), and 0x00000414 in the UTF-32.

Originally, Unicode specified 12,793 different character symbols. But soon it became clear that this number of symbols is far from enough, so Unicode has been significantly extended. The last version of Unicode is 13.0. Unicode 13.0 specifies a total of different 143,859 characters.

Currently, the most popular Unicode encoding is **UTF-8**. It is especially popular on the Internet. This character encoding uses 1 to 4 bytes per symbol. Therefore, it is a variable-width encoding, in contrast to ASCII. However, UTF-8 is totally backward-compatible with ASCII, which is incredibly useful. In UTF-8, the ASCII symbols are unchanged.

In UTF-8, symbols that tend to occur more frequently – for example, the ASCII symbols – are encoded with fewer bytes. This allows for using less memory for such encoding, and it is one of the major advantages of this encoding.

In UTF-8, the leading zero indicates that the character is encoded with only 1 byte. Obviously, one-byte encoded characters in UTF-8 coincide with the ASCII characters.

Other than one-byte encoded characters begin with the control bits. Two-byte characters begin with **110**. Three-byte characters begin with **1110**. And four-byte characters begin with **11110**. The following bytes of such characters also begin with control bits. In this case, the control bits are the same: **10**.

For example, these are encodings of two-byte, three-byte, and four-byte characters, in which the control bits are in red:

11010101 10100101
11101010 10101010 10011110
11110100 10101011 10000010 10101010

UTF-16 is another popular Unicode character encoding. In UTF-16, codes symbols consist either of 16 bits (2 bytes) or 32 bits (4 bytes). So, like UTF-8, UTF-16 is also a variable-width character encoding. UTF-16 is totally backward-compatible with the Unicode encoding UCS-2, which is currently obsolete, but was very popular not long ago.

For encoding texts that mostly consist of ASCII symbols, it is advisable to use UTF-8 because it requires less computer memory. But some other types of text may be more efficiently encoded using UTF-16.

Currently, **UTF-32**, also known as **UCS-4**, is a less popular encoding than UTF-8 or UTF-16 encodings. UTF-32 is a fixed-length encoding, which uses 32 bits (4 bytes) per symbol. In UTF-32, codes for the ASCII symbols are zero-extended. For example, the ASCII code for the “A” symbol is

01000001, but in UTF-32 it transforms into **00000000 00000000 00000000 01000001**.

The obvious and most important disadvantage of UTF-32 is that using this encoding requires more computer memory than using UTF-8 or UTF-16. However, the process of decoding UTF-32 is oftentimes simpler and consequently requires less processor time than decoding UTF-8 or UTF-16.

Introduction to Representing Images

In general, there are two major methodologies for representing images: raster graphics and vector graphics.

Raster graphics is mainly used for representing analog images such as photographs. In this methodology, images are represented as collections of dots, called *pixels*, short for “picture element”. A collection of pixels is called a *bit map*.

In raster graphics, a very important image characteristic is *resolution*. The resolution is the number of pixels per unit of image area or length, for example, ppi or pixels per inch. The higher the resolution, the smaller the pixels the images have, and the better the quality of the image.

There are many different methods for encoding pixels. The simplest method is used for representing monochrome images, such as some black and white pictures. In this method, each pixel can have only two different values: a value for a white color and another one for a black color. So, this encoding requires only one bit per pixel.

Other pixel-based encoding methods are used for color images. One of such methods is the **RGB encoding**.

The white light that we all see is actually a mixture of different color lights. And we can make pretty much any color by mixing different amounts of some primary colors.

The RGB encoding is based on 3 *primary colors*: red, green, and blue. This color encoding uses 3 bytes per pixel – one byte per primary color. Each byte encodes the *intensity* of one primary color.

As you probably remember, a byte can hold 256 different values, from 0 to 255.

0...255, 0...255, 0...255

RGB is an *additive* color system, which means that in this color system we add colors to produce another color.

Since we use 3 bytes per pixel in RGB, there are 16 777 216 possible colors ($2^8 * 2^8 * 2^8$) this encoding can produce.

Color depth is a measure of bits used to encode a pixel. The color depth of RGB images is 24. The bigger the color depth the more memory is needed for image encoding, but the better image quality can be achieved.

Usually, we don't use binary or decimal values when we deal with RGB pixels. Instead, *hex triplets* are more commonly used in a format #rrggbb, where rr is a hexadecimal value for red, gg – for green, and bb – for blue.

Some examples hex triplets for different RGB images:

#FFFFFF – white color

#FF0000 – red color

#00FF00 – green color

#0000FF – blue color

#FFFF00 – yellow color (mixing red and green produces yellow)

#FF00FF – magenta color (mixing red and blue produces magenta)

#00FFFF – cyan color (mixing green and blue produces cyan)

#000000 – black color

Given the size of an image in pixels, we can easily calculate the size of the image in bytes.

For example, we have an RGB picture of 100x200 pixels. This gives 20000 pixels in total. Then we just multiply 20000 by 3 bytes per each

pixel. So, the size of this image without any file metadata is 60000 bytes or 60 KB or 58,6 KiB.

This whole scheme we've just discussed, where each pixel in an image is encoded separately is called the **true-color scheme**. As you probably guess, this is not a very efficient representation in terms of memory usage.

But there is an alternative to the true-color scheme. It is called the **indexed color or palette color scheme.** In this method, an image can have only a fraction, usually small, of 2^{64} possible colors that can be encoded by the true-color RGB encoding. It's usually 256 colors. Each file, in this method, has a table of colors known as a *palette*. And each pixel is basically a reference to one color from the palette. If we use, say 256 colors, we need only 8 bits per reference. This method allows us to considerably reduce the size of the image while using the same RGB encoding.

It is interesting that the RGB encoding is somewhat similar to how human eyes perceive colors. The point is that our eyes have different types of photoreceptor cells, which respond to the three colors: red, green, and blue.

In addition to the RGB encoding, there are other methods of encoding pixels. As we have seen, RGB is an additive color system. But, there are also subtractive color systems, in which different colors are produced by removing certain colors from light rather than adding colors as in additive systems. One of such subtractive color systems is **CMY** that is based on other three primary colors: cyan, magenta, and yellow.

In some other pixel-based image system encoding, we use additional information to encode the transparency of each pixel. Usually, this component is called *alpha*.

There are many file formats for raster graphic encoding, but some of them are more common and can be even considered as standards.

One such commonly used image file format is **JPEG** (Joint Photographic Experts Group). It uses the True-Color RGB scheme and compasses the image. It uses *lossy compression* also known as *irreversible compression*. Lossy compression implies that we lose some information when we

compress the data. In this format, the degree of compression can be adjusted, which allows the users to adjust the trade-off between the image quality and the size of the files. JPEG is probably the most commonly used format of raster graphics.

Another commonly-used raster image format is **GIF** (Graphic Interchange Format). It uses the indexed color scheme for image encoding. It also uses data compression, but the compression is *lossless*, as opposed to the JPEG format. Also, GIF can support animation. This file format is more common on the Web, especially on social media platforms.

Representing images as collections of bit maps has a very important disadvantage. If we use pixels, we cannot rescale the images to different sizes, without losing image quality. Sometimes this loss in quality is substantial and even unacceptable. Because of this, there is another methodology for computer image representation – vector graphics.

Unlike raster graphics, vector graphics does not use bit maps to encode all pixels. Instead in vector graphics, an image is represented as a collection of geometric shapes, such as lines, squares, and curves, that can be encoded using techniques of analytic geometry. For example, a line can be represented as the coordinates of its endpoints plus some other additional information.

Vector graphics is not commonly used for storing analog images, such as photos. Vector graphics is popular in many drawing programs, word processing systems to represent different fonts of text, computer-aided design (CAD) systems, and some other applications. Usually, computer drawing programs use a combination of vector and raster graphics and can convert one type of graphics into another.

Introduction to Representing Sound

As simple as it may seem, a sound is a pressure wave that can be transmitted through different states of matter. Usually, we perceive sound as a wave in the air.

In the case of sound, the amplitude of the signal is loudness. Another very important characteristic of sound, as of any other type of wave, is frequency. Frequency is the rate of amplitude spikes (cycles) per second, and it is measured in hertz (Hz). One hertz is one cycle per second.

Humans can hear sound signals with frequencies approximately between 20 Hz and 20 kHz. Some other species perceive sounds that have other frequency ranges. For example, dogs can hear sounds with frequency up to 40 kHz and even higher.

Sound is an example of analog signals. An analog signal doesn't have defined ranges. What also very important is that an analog signal is continuous. We can't store continuous signals in computer memory because the memory is finite.

For storing and processing an analog signal in a computer, we must first convert it to digital information. This is done by the processes called *sampling* and *quantization*.

Sampling is the process in which we select only a finite number of time points on the analog signal graph, measure their values, and record them.

How many samples we need in each second in order to reproduce the sound well depends on the sound frequency. The lower the frequency of a sound, the fewer samples we need to take. But for the general case, it was estimated that approximately 40 000 samples per second (40 kHz) is good enough to reproduce an audio signal for humans. In fact, in information theory, there is the *Nyquist–Shannon sampling theorem* that determines how often we need to sample an analog signal to represent it in digital form with good quality.

Also, because of the fact that computer memory is finite and an amplitude value is a real number, we can't store each and every possible value of amplitude in computer memory. For this reason, we use the process, which is similar to sampling, that is called *quantization*. Quantization is the process in which the value of a sample is rounded to the closest integer value.

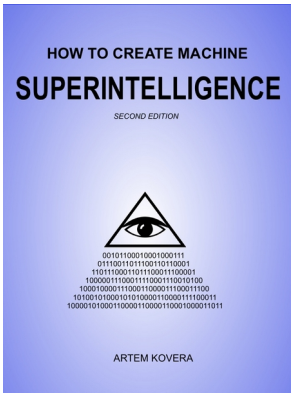
There are different schemes of encoding those values. Usually, the data obtained from each sample are represented in 16, 24, or even 32 bits. This number of bits needed to encode one value of a sample is sometimes referred to as the *bit depth*. Obviously, the greater the bit depth, the more memory is needed to record the sound, and the higher the quality of the sound can be achieved.

If we take *S* samples per second and denote the bit depth as *B*, we need to store *S* x *B* bits per second. For example, if the sampling rate is 40000 and the bit depth is 16, we need 40000 x 16 bits per second or 80 kB per second without compression.

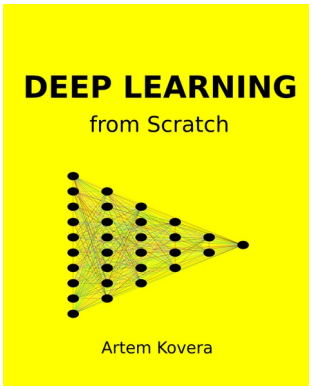
There are multiple file formats for sound storing. Some of them can be considered as standards. One of such standards is **MP3** (short for **MPEG Layer 3**). This standard uses 44 100 samples per second and 16 bits per sample. It also uses lossy compression that discards some information the human ear cannot hear.

My Books on Artificial Intelligence and Machine Learning

[How to Create Machine Superintelligence: A Quick Journey through Classical/Quantum Computing, Artificial Intelligence, Machine Learning, and Neural Networks]



[Deep Learning from Scratch: From Basics to Building Neural Networks with Keras]



[Machine Learning with Clustering: A Visual Guide with Examples in Python]